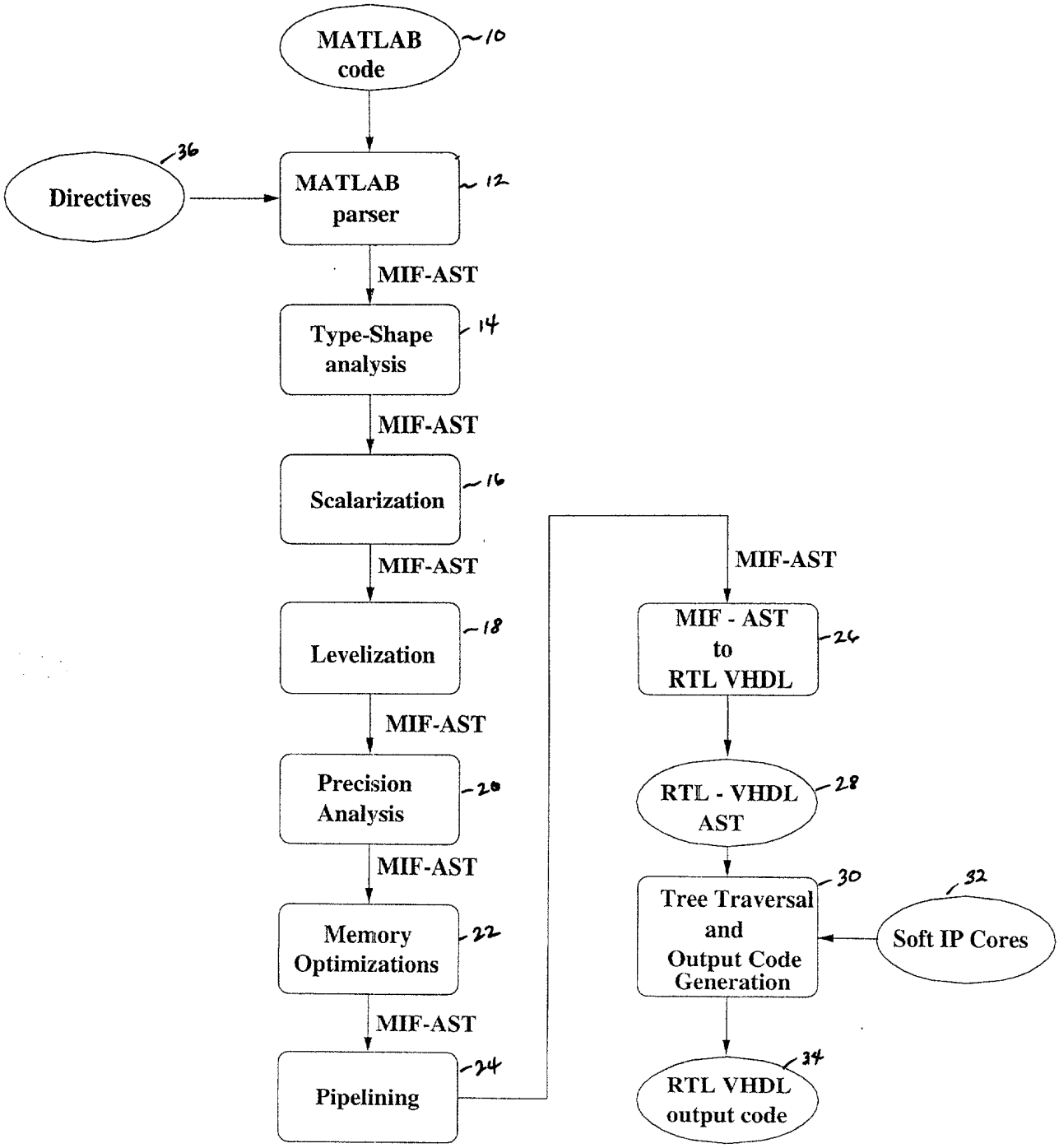
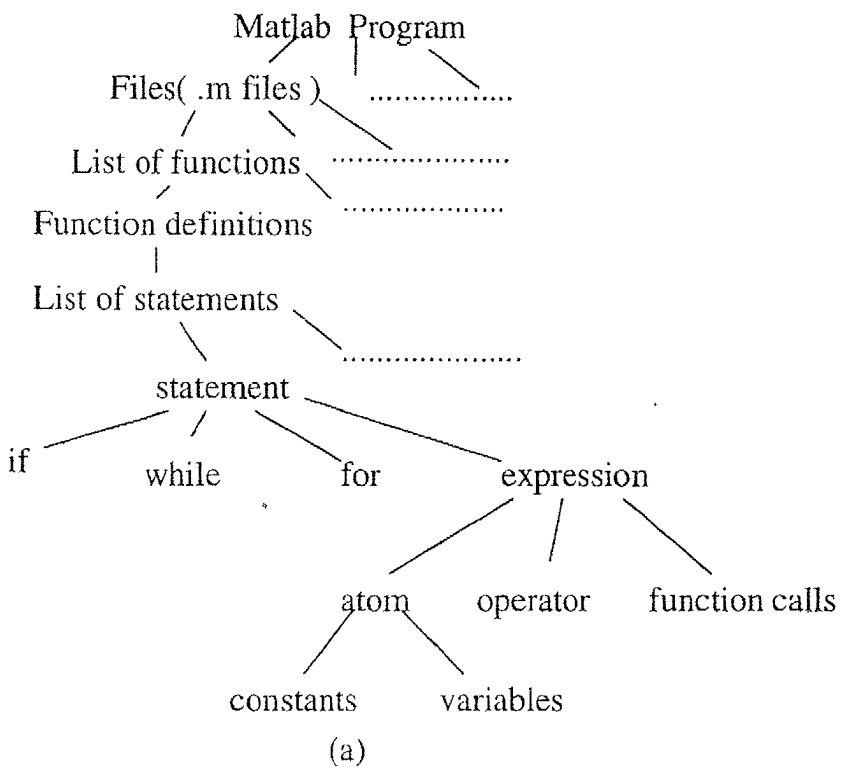


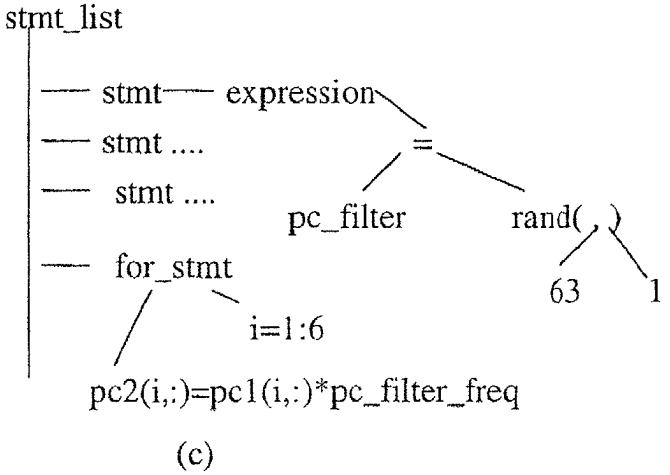
Figure 1: Synthesis flow.



(a)

```
pc_filter = rand(63,1);
pc_filter_time = zeros9512,1);
pc_filter_frq = fft(pc_filter_time);
for i = 1:6
  pc2(i,:) = pc1(i,:)*pc_filter_freq
end;
```

(b)



(c)

Figure 2: Abstract Syntax Tree: (a) The hierarchy captured by the formal grammar (b) A sample code snippet (c) Abridged syntax tree for the code snippet.

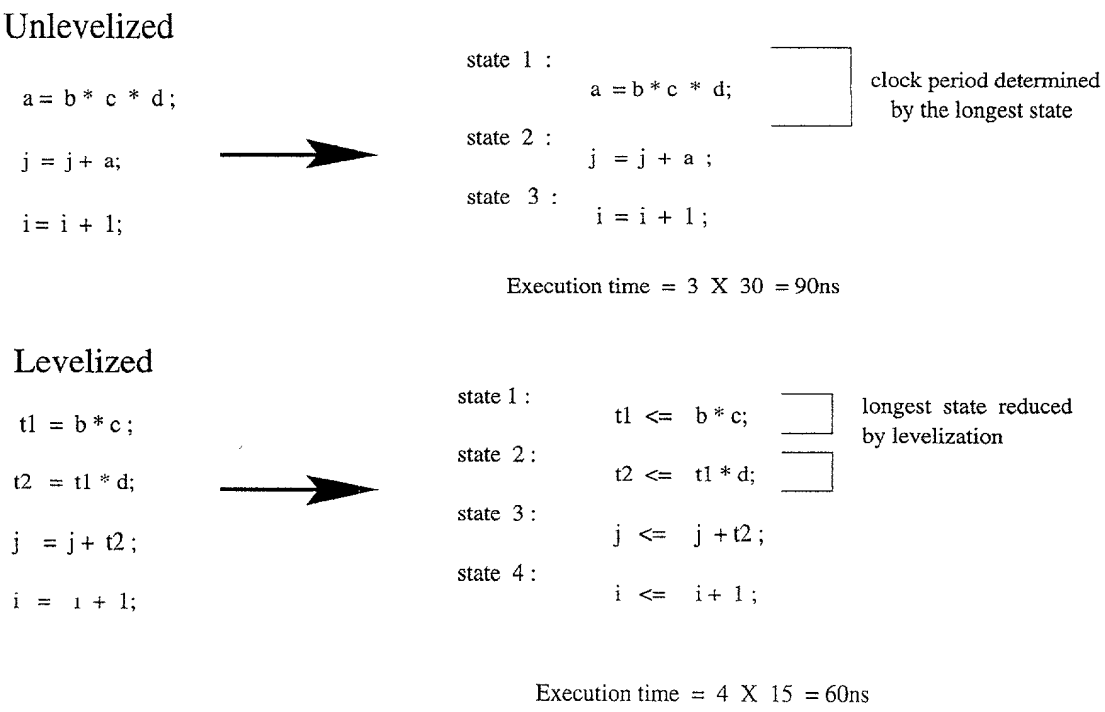


Figure 3: Levelization improves clock period.

```
a = 1 ;
b = 1 ;
c = a + b ;
c = c + 1 ;
```

(a)

```
when state 1 =>  a <= 1 ;
                 next_state <= state 2 ;
when state 2 =>  b <= 1 ;
                 next_state <= state 3 ;
when state 3 =>  c <= a + b ;
                 next_state <= state 4 ;
when state 4 =>  c <= c + 1 ;
```

(b)

Figure 4: (a) Simple MATLAB statements. (b) State machine that steps through the statements sequentially.

```
if( x )
  a = b + 1;
else
  a = b - 2;
end;
c = a + 1;
```

(a)

```
when state 1 =>  if( x ) then
                   next_state <= state 2 ;
                 else
                   next_state <= state 3 ;
when state 2 =>  a <= b + 1 ;
                   next_state <= state 4 ;
when state 3 =>  a <= b - 2 ;
                   next_state <= state 4;
when state 4 =>
                 c <= a + 1 ;
```

(b)

Figure 5: (a) Conditional MATLAB code (b) State machine for Conditional Code.

```
for i = 1 : 256
    a = a + i ;
end ;
```

(a)

```
when state 1 =>  i <= 1 ;
                 next_state <= state 2 ;
when state 2 =>  if( i <= 256 ) then
                  next_state <= state 3 ;
                else
                  next_state <= state 5;
                 endif ;
when state 3 =>   a <= a + i ;
                 next_state <= state 4 ;
when state 4 =>   i <= i + 1 ;
                 next_state <= state 2 ;
when state 5 =>  [ next statement after loop ]
```

(b)

Figure 6: (a) A MATLAB *for* loop (b) State machine for *for* loop.

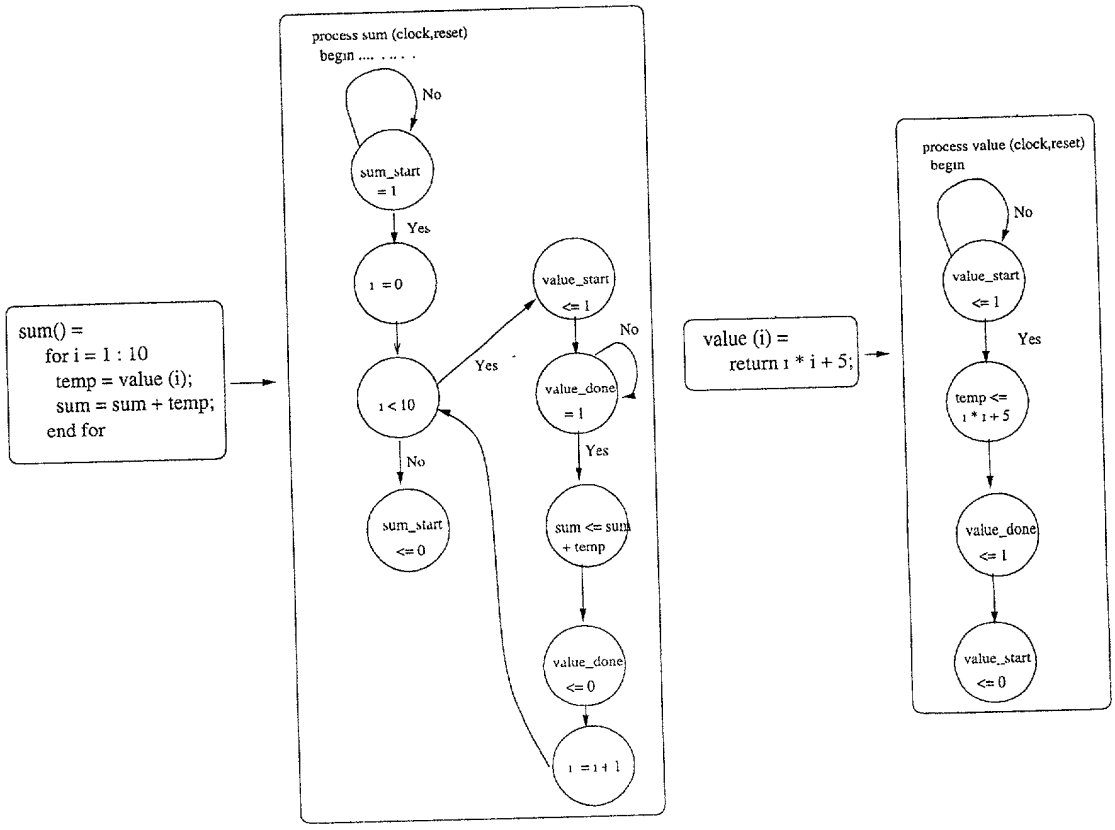


Figure 7: State machine representation of a function call in MATLAB

```
a[ i + 1 ] = b + c ;
        |
   Levelization
        ⇓

t1 = b + c ;
t2 = i + 1;
a[ t2 ] = t1 ;
```

(a)

```
when state 1 => t1  <=  b + c ;
                next_state <= state 2 ;
when state 2 => t2  <=  i  +  1 ;
                next_state <= state 3 ;
when state 3 => mem_request  <=  '0' ;
                mem_write_enable <= '0' ;
                next_state <= state 4 ;
when state 4 => mem_address <= Base_a  + t2 ;
                mem_data_out <= t1 ;
                next_state <= state 5 ;
when state 5 =>  if( mem_grant = '0' ) then
                    next_state <= state 6;
                 else
                    next_state <= state 4 ;
                 endif;
```

(b)

Figure 8: (a) Array statement in MATLAB and its levelization (b) VHDL corresponding to the MATLAB code. The signals *mem_request*, *mem_data_out*, *mem_grant*, *mem_write_enable* and their particular states and assignments are specified in an external file read by the compiler. *Base_a* is a constant denoting the starting address of the array *a* in memory.

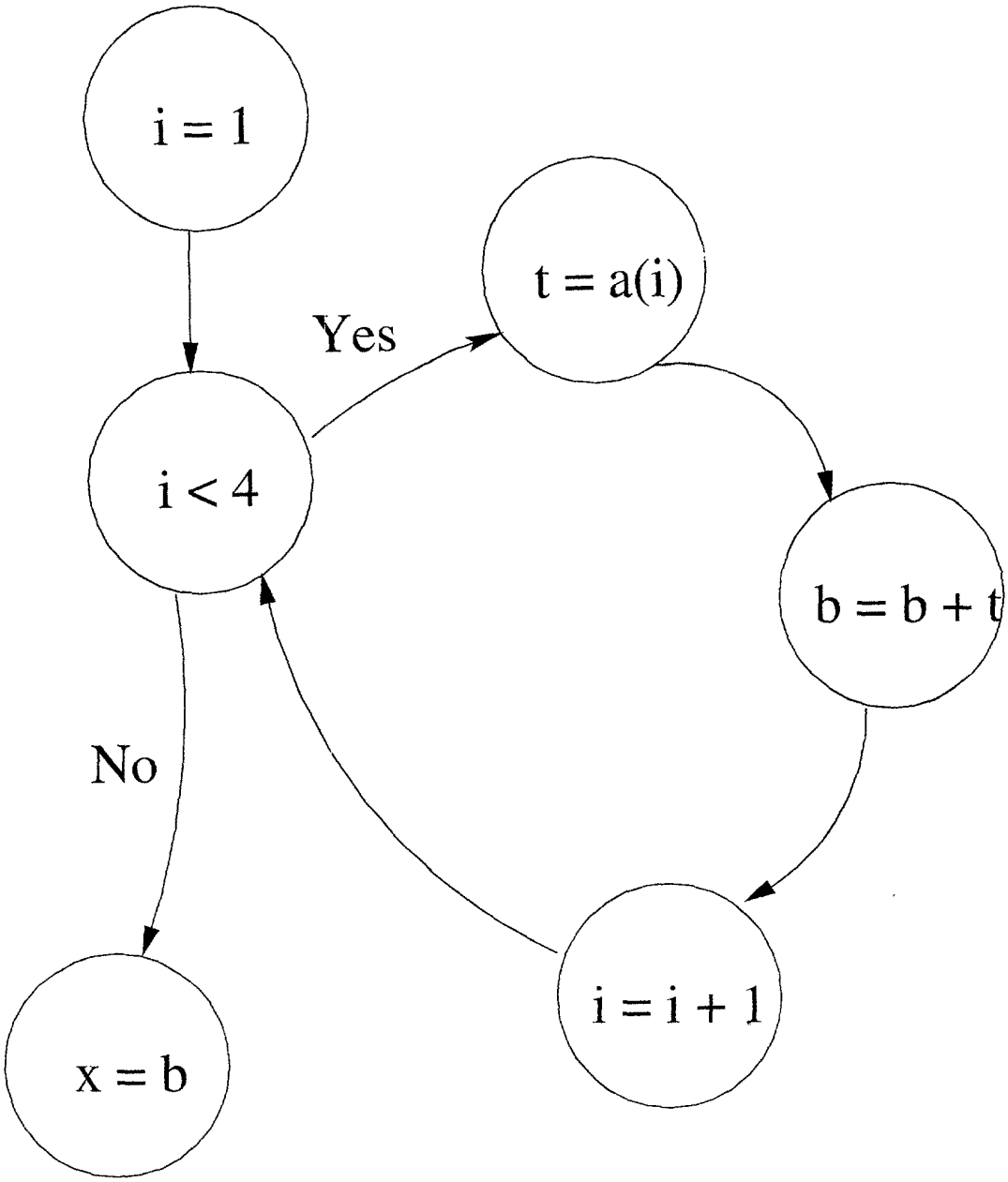


Figure 9: FSM Representation of a code section

Algorithm 1 architecture ...
process ...

```
if reset = '1' then
    ...
elsif rising_edge(clock)
    case control is
        when state1 ⇒
            i := 1;
            control ⇐ state2;
        when state2 ⇒
            if (i < 4) then
            ....
            else
            ....
            endif;
        when state3 ⇒
            t := a(i);
            control ⇐ state4;
        when state4 ⇒
            b := b + t;
            control ⇐ state5;
        when state5 ⇒
            i := i + 1;
            control ⇐ state2;
```

Figure 10: VHDL code generated for the above FSM

over manual design or design at a low level of algorithm description (e.g., directly in VHDL).

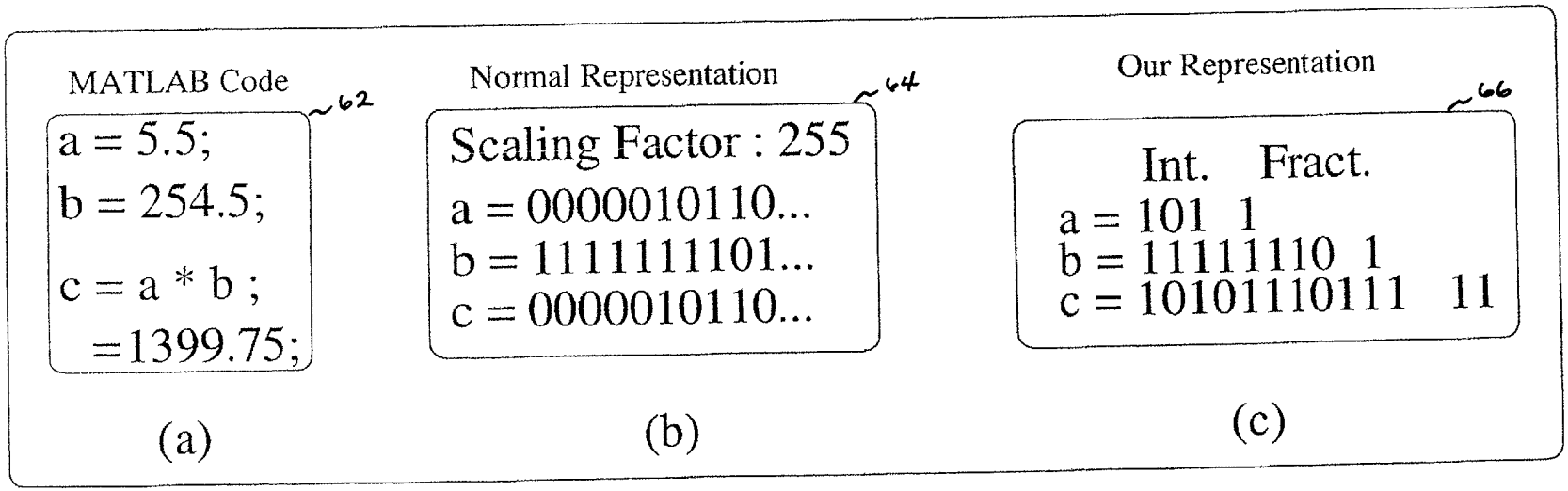


Figure 11: Representation of Real Variables

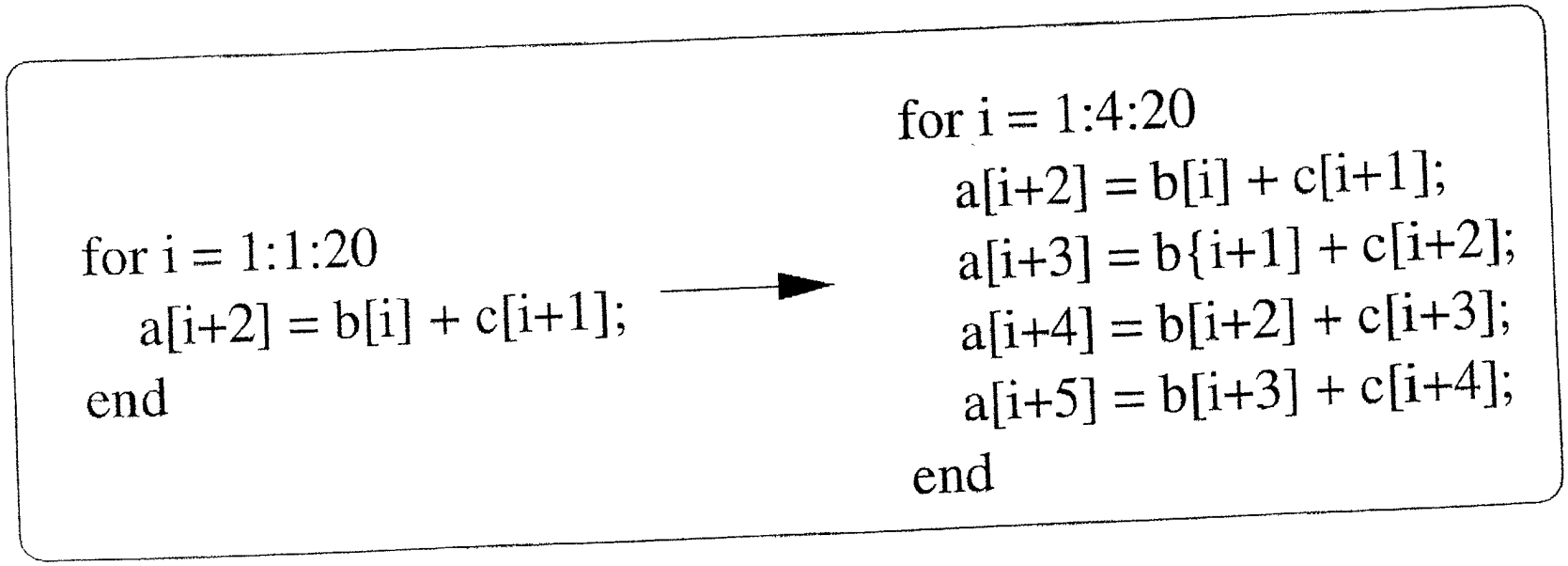


Figure 12: Example showing loop unrolled for Memory Packing

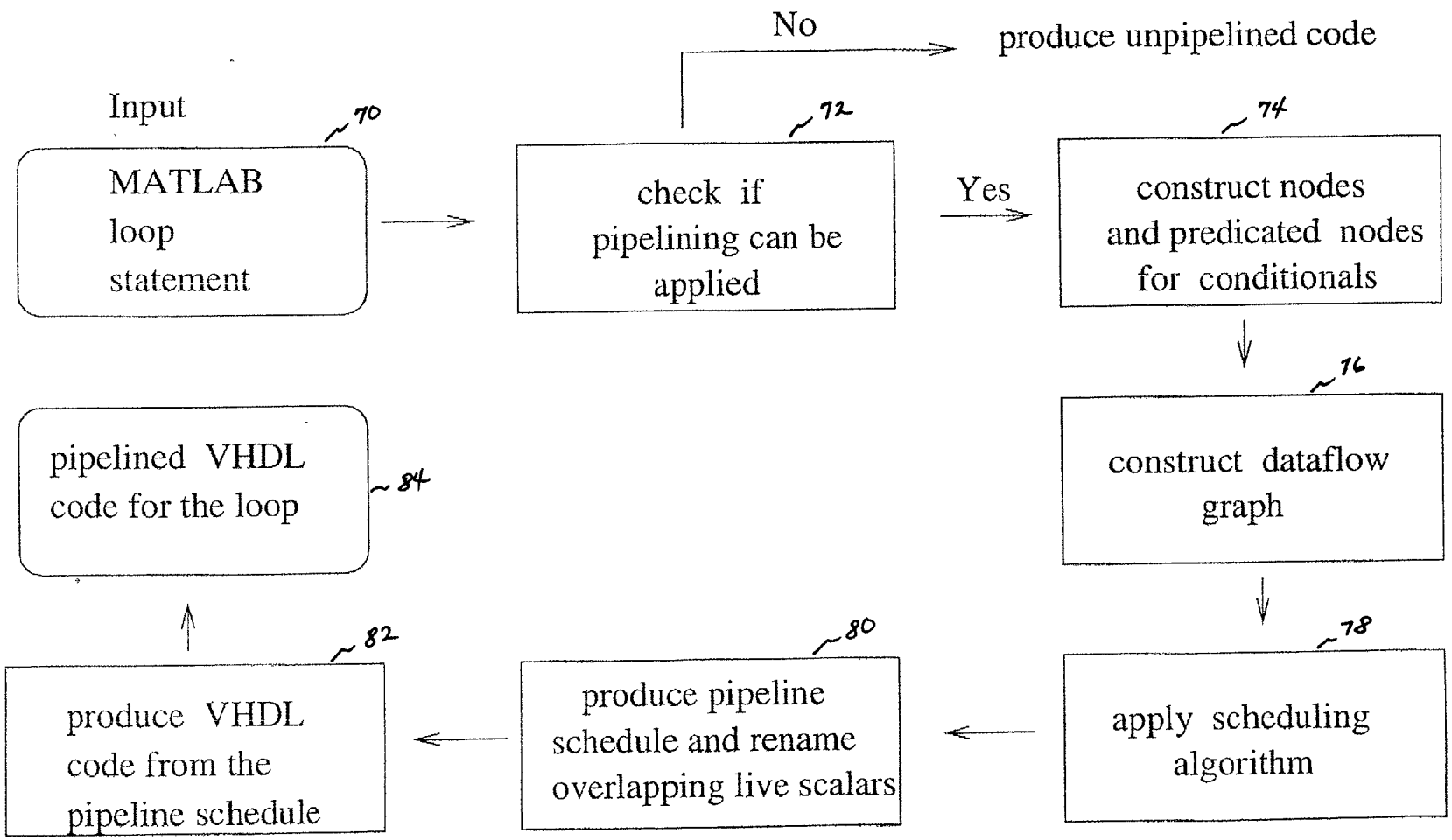
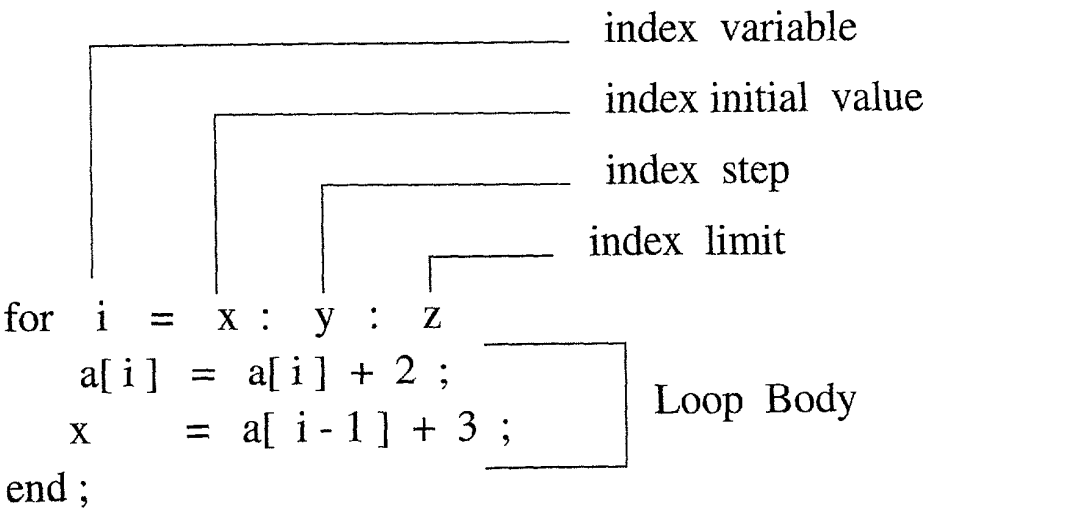
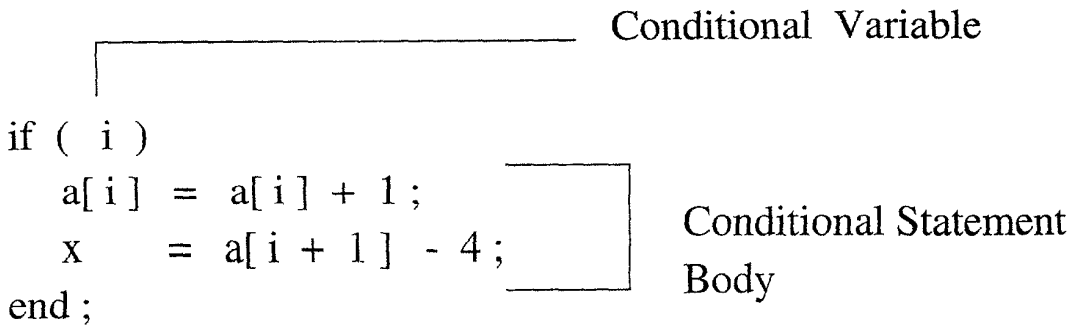


Figure 13: Overall framework for pipelining optimizations



Loop Statement

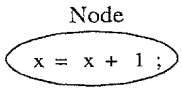


Conditional Statement

Figure 14: Illustration of Terms used in pipelining framework

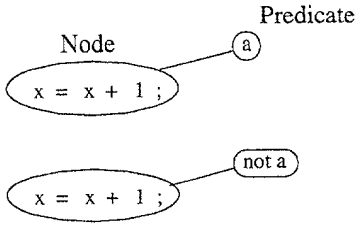
Simple Statements

```
x = x + 1 ;
```



Conditional Statements

```
if( a )
  x = x + 1 ;
else
  x = b * 2 ;
end ,
```



Nested Conditional Statements

```
if( a )
   if( t )
      x = x + 1 ;
   end ,
else
   if( k )
      x = 2 * y ;
   else
      g = g + 1 ;
   end ;
end;
```

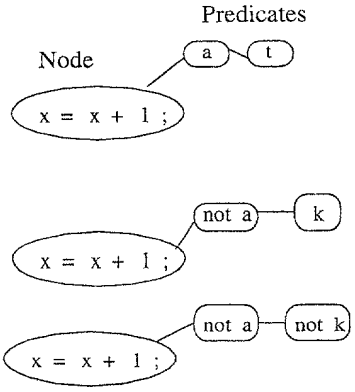


Figure 15: Illustration of Construction of nodes from MATLAB statements

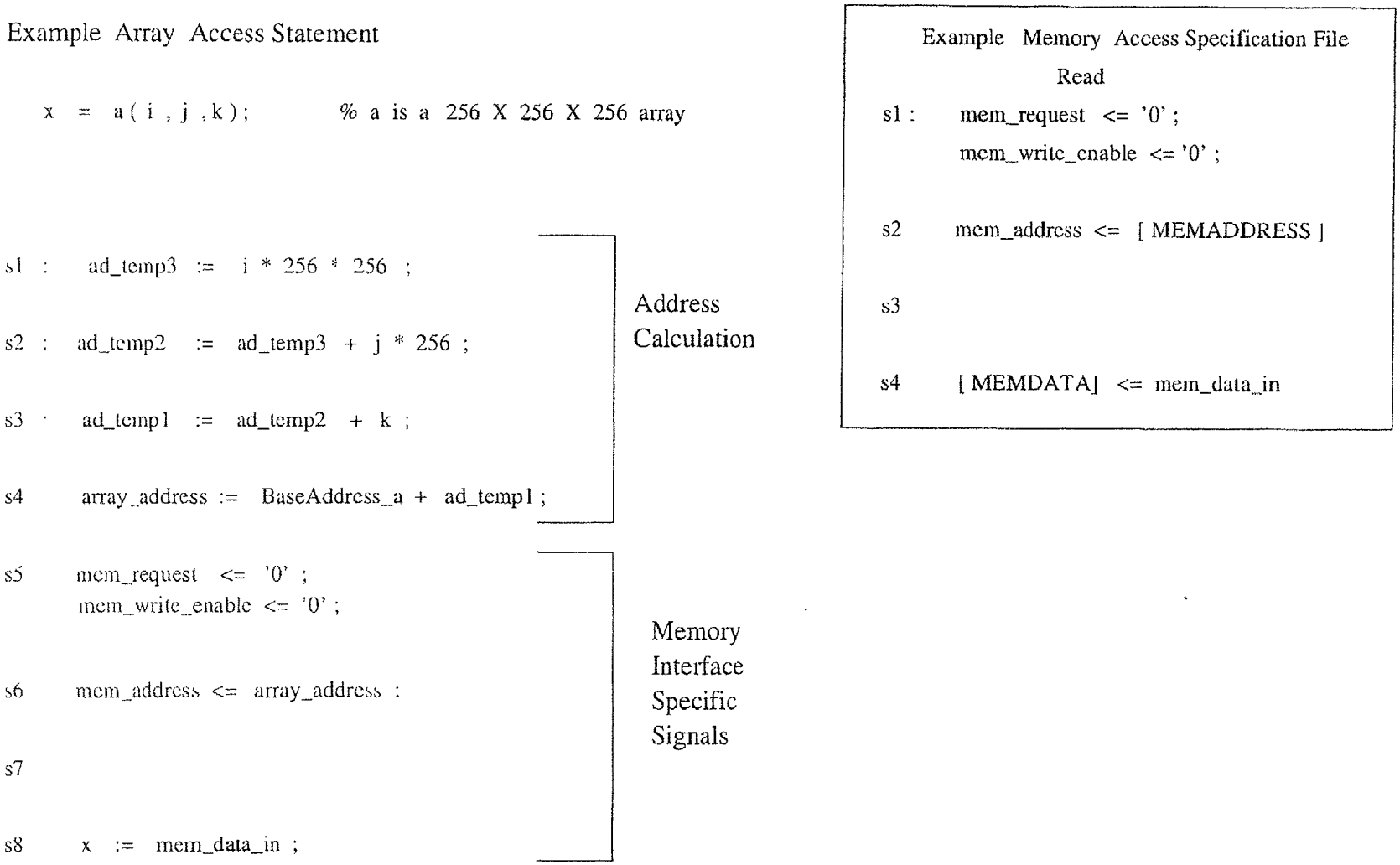


Figure 16: Example of node construction for array access statements

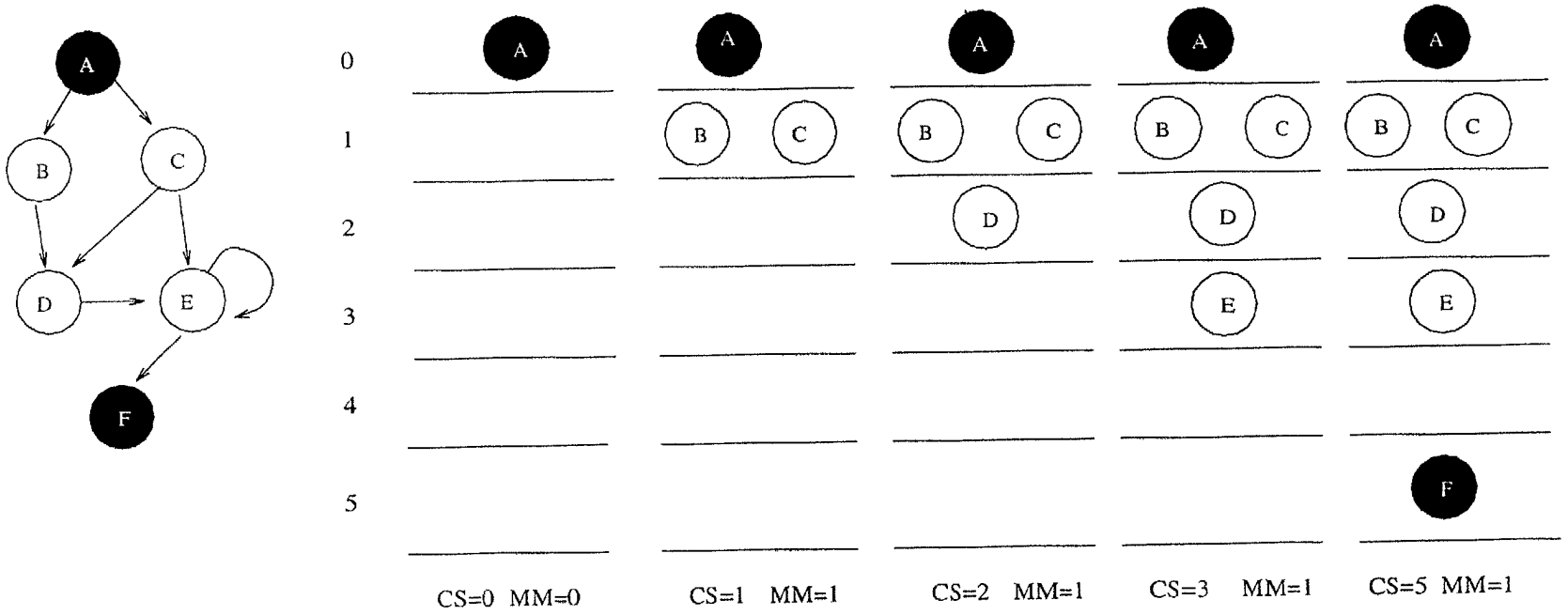


Figure 17: An illustration of pipeline method

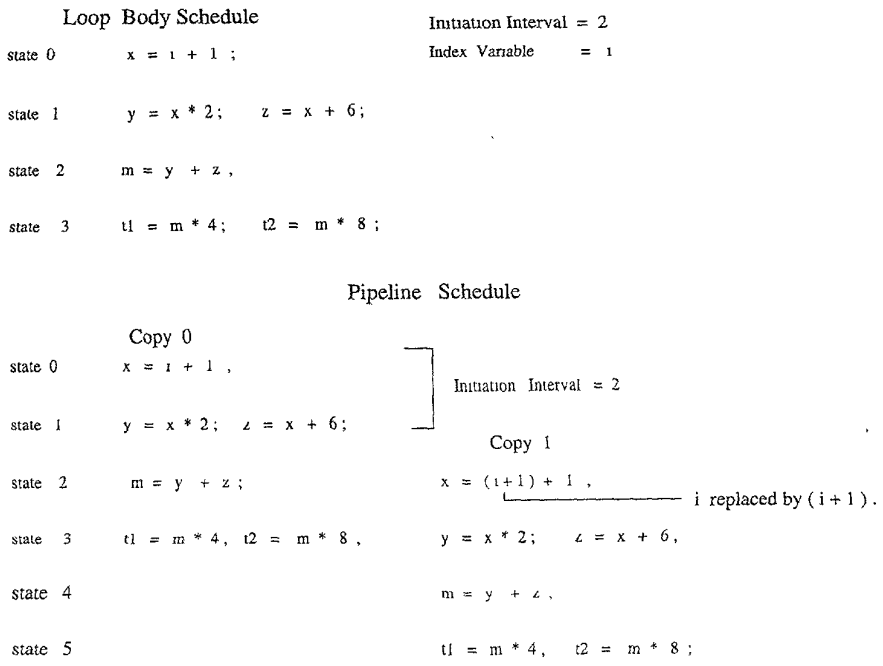


Figure 18: Construction of pipeline schedule from loop body schedule

Loop Body Schedule

```
s1   x = i + 1 ;
s2   y = i * 2 ,
s3   k = x + 3 ,
```

Pipeline Schedule

```
          Copy 0              Copy 1                    Copy 2
s1   x = i + 1 ;  x live range
s2   y = i * 2 ;    x = i + 1 ;  x live range
s3   k = x + 3 ;    y = i * 2 ;    x = i + 1 ;  x live range
s4                  k = x + 3 ;    y = i * 2 ;
s5                                 k = x + 3 ;
```

Pipeline Schedule with Overlapping Live Scalars Renamed

```
                Copy 0
s1   case mod_var is
       when 0 :  x0 = i + 1;
       when 1 :  x1 = i + 1,
       when 2 :  x2 = i + 1;
     end case ;

                                  Copy 1
s2   y = i * 2 ;                  case mod_var is
                                    when 0 :  x1 = i + 1;
                                    when 1 :  x2 = i + 1;
                                    when 2    x0 = i + 1,
                                  end case ;

                                                      Copy 2
s3   case mod_var is              y = i * 2 ;         case mod_var is
       when 0   k = x0 + 3;                             when 0 :   x2 = i + 1;
       when 1 : k = x1 + 3,                             when 1     x0 = i + 1;
       when 2 : k = x2 + 3;                             when 2 .   x1 = i + 1;
     end case ,                                       end case ;

s4                                case mod_var is     y = i * 2 ;
                                    when 0   k = x1 + 3,
                                    when 1 . k = x2 + 3,
                                    when 2 · k = x0 + 3,
                                  end case ,

s5                                                    case mod_var is
                                                        when 0   k = x2 + 3;
                                                        when 1 : k = x0 + 3,
                                                        when 2   k = x1 + 3;
                                                      end case ,
```

Figure 19: Renaming of scalars with live overlapping ranges in the pipeline schedule

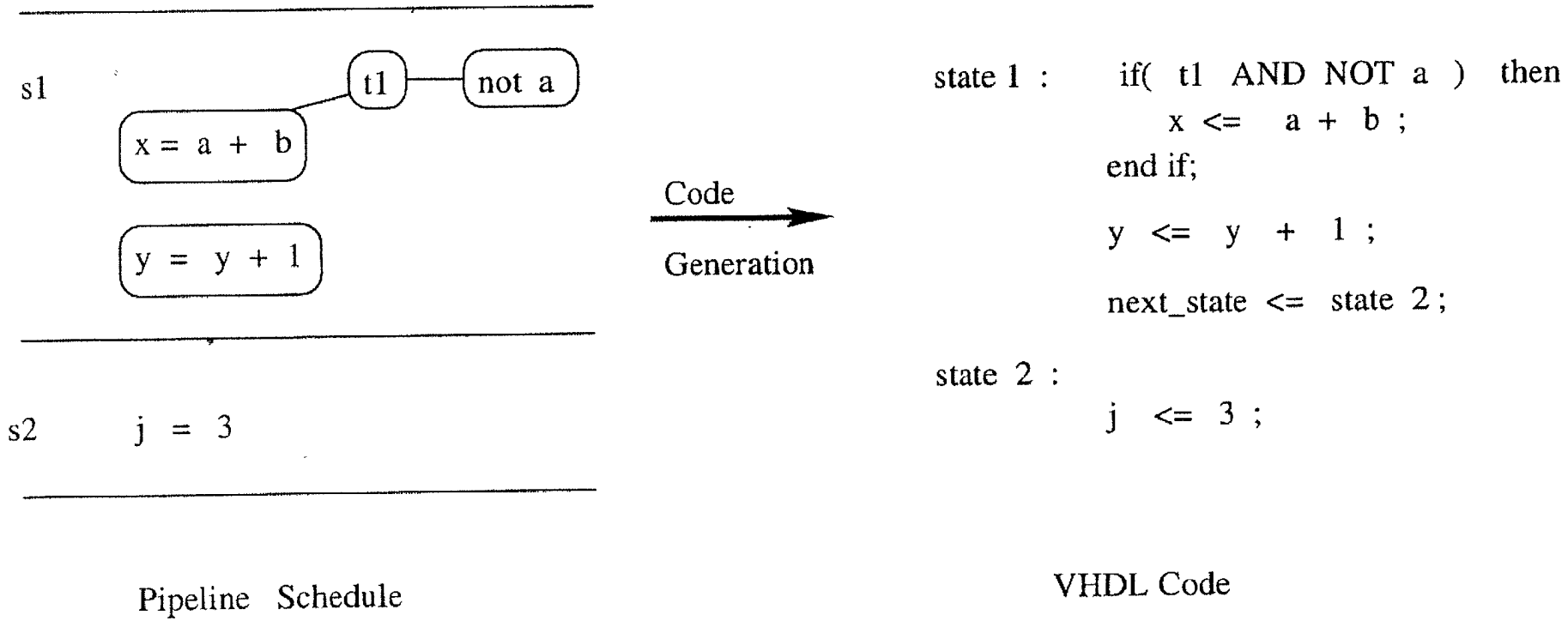


Figure 20: VHDL Code generation Illustration